Chapter TWENTY-TWO
Time Zones and Daylight Savings

Exam Objectives

Work with dates and times across time zones and manage changes resulting from daylight savings including Format date and times values.

Core time zone classes

Before Java 8, if we wanted to work with time zone information, we have to use the class `java.util.TimeZone` . Now with the new Date/Time API, there'are new and better options.

They are:

ZoneID
Represents the ID of time zone. For example, *Asia/Tokyo*.

ZoneOffset
Represents a time zone offset. It's a subclass of ZoneId. For example, *-06:00*.

ZonedDateTime
Represents a date/time with time zone information. For example, *2015-08-30T20:05:12.463-05:00[America/Mexico_City]*.

OffsetDateTime
Represents a date/time with an offset from UTC/Greenwich. For example, *2015-08-30T20:05:12.463-05:00*.

OffsetTime
Represents a time with an offset from UTC/Greenwich. For example, *20:05:12.463-05:00*.

Just like the classes of the previous chapter, these one are located in the `java.time` package and are immutable.

ZoneId and ZoneOffset classes

The world is divided into time zones in which the same standard time is kept. By convention, a time zone is expressed as the number of hours different from the Coordinated Universal Time (*UTC*). Since the Greenwich Mean Time (*GMT*) and the

Zulu time (*Z*), used in military, have no offset from *UTC*, they're often used as synonyms.

Java uses the Internet Assigned Numbers Authority (IANA) database of time zones, which keeps a record of all known time zones around the world and is updated many times per year.

Each time zone has an ID, represented by the class `java.time.ZoneId`. There are three types of ID:

The first type just states the offset from UTC/GMT time. They are represented by the class `ZoneOffset` and they consist of digits starting with + or -, for example, *+02:00*.

The second type also states the offset from UTC/GMT time, but with one of the following prefixes: *UTC*, *GMT* and *UT*, for example, *UTC+11:00*. They are also represented by the class `ZoneOffset`.

The third type is region based. These IDs have the format *area/city*, for example, *Europe/London*.

You can get all the available zone IDs with the `static` method:

```
Set<String> getAvailableZoneIds()
```

For example, to print them in the console:

```
ZoneId.getAvailableZoneIds().stream().forEach(System.out::println);
```

To get the zone ID of your system use the `static` method:

```
ZoneId.systemDefault()
```

Under the cover, it uses `java.util.TimeZone.getDefault()` to find the default time zone and converts it to a `ZoneId`.

If you want to create a specific `ZoneId` object use the method `of()`:

```
ZoneId singaporeZoneId = ZoneId.of("Asia/Singapore");
```

This method parses the ID producing a `ZoneOffset` or a `ZoneRegion` (both extend from `ZoneId`).

Actually, the above line produces a `ZoneRegion`. A `ZoneOffset` is returned if for example, ID is *Z*, or starts with + or -. For example:

```
ZoneId zoneId = ZoneId.of("Z"); // Z represents the zone ID for UTC
ZoneId zoneId = ZoneId.of("-2"); // -02:00
```

The rules for this method are:

- If the zone ID equals Z, the result is `ZoneOffset.UTC`. Any other letter will throw an exception.
- If the zone ID starts with + or -, the ID is parsed as a `ZoneOffset` using `ZoneOffset.of(String)`.
- If the zone ID equals `GMT`, `UTC` or `UT` then the result is a `ZoneId` with the same ID and rules equivalent to `ZoneOffset.UTC`.

- If the zone ID starts with `UTC+`, `UTC-`, `GMT+`, `GMT-`, `UT+` or `UT-` then the ID is split in two, with a two or three letter prefix and a suffix starting with the sign. The suffix is parsed as a `ZoneOffset`. The result will be a `ZoneId` with the specified prefix and the normalized offset ID.
- All other IDs are parsed as region-based zone IDs. If the format is invalid (it has to match the expression `[A-Za-z][A-Za-z0-9~/._+-]+)` or is not found, an exception is thrown.

Remember that a `ZoneOffset` represents an offset, generally from UTC. This class has a lot more constructors than `ZoneId`:

```
// The offset must be in the range of -18 to +18
ZoneOffset offsetHours = ZoneOffset.ofHours(1);
// The range is -18 to +18 for hours and 0 to ± 59 for minutes
// If the hours are negative, the minutes must be negative or zero
ZoneOffset offsetHrMin = ZoneOffset.ofHoursMinutes(1, 30);
// The range is -18 to +18 for hours and 0 to ± 59 for mins and secs]
// If the hours are negative, mins and secs must be negative or zero
ZoneOffset offsetHrMinSe = ZoneOffset.ofHoursMinutesSeconds(1,30,0);
// The offset must be in the range -18:00 to +18:00
// Which corresponds to -64800 to +64800
ZoneOffset offsetTotalSeconds = ZoneOffset.ofTotalSeconds(3600);
// The range must be from +18:00 to -18:00
ZoneOffset offset = ZoneOffset.of("+01:30:00");
```

The formats accepted by the `of()` method are:

- *Z (for UTC)*
- *+h*
- *+hh*
- *+hh:mm*
- *-hh:mm*
- *+hhmm*
- *-hhmm*
- *+hh:mm:ss*
- *-hh:mm:ss*
- *+hhmmss*
- *-hhmmss*

If you pass an invalid format or an out-of-range value to any of these methods, an exception is thrown.

To get the value of the offset, you can use:

```
// Gets the offset as int
int offsetInt = offset.get(ChronoField.OFFSET_SECONDS);
// Gets the offset as long
long offsetLong= offset.getLong(ChronoField.OFFSET_SECONDS);
// Gets the offset in seconds
int offsetSeconds = offset.getTotalSeconds();
```

`ChronoField.OFFSET_SECONDS` is the only accepted value of `ChronoField`, so the three statements above return the same result. Other values throw an exception.

Anyway, once you have a `ZoneId` object, you can use it to create a `ZonedDateTime` instance.

A ZonedDateTime object

```
    Date                        Offset
2015-08-31 T08:45:20.000 +02:00[Africa/Cairo]
           Time                 Time zone
```

ZonedDateTime class

A `java.time.ZonedDateTime` object represents a point in time relative to a time zone.

A `ZonedDateTime` object has three parts:

- A date
- A time
- A time zone

Which means that it stores all date and time fields, to a precision of nanoseconds, and a time zone with a zone offset.

So once you have a `ZoneId` object, you can combine it with a `LocalDate`, a `LocalDateTime`, or an `Instant`, to transform it into `ZonedDateTime`:

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

LocalDate date = LocalDate.of(2010, 7, 3);
ZonedDateTime zonedDate = date.atStartOfDay(australiaZone);

LocalDateTime dateTime = LocalDateTime.of(2010, 7, 3, 9, 0);
ZonedDateTime zonedDateTime = dateTime.atZone(australiaZone);

Instant instant = Instant.now();
ZonedDateTime zonedInstant = instant.atZone(australiaZone);
```

Or using the `of` method:

```
ZonedDateTime zonedDateTime2 =
    ZonedDateTime.of(LocalDate.now(), LocalTime.now(), australiaZone);
ZonedDateTime zonedDateTime3 =
    ZonedDateTime.of(LocalDateTime.now(), australiaZone);
ZonedDateTime zonedDateTime4 =
    ZonedDateTime.ofInstant(Instant.now(), australiaZone);
// year, month, day, hours, minutes, seconds, nanoseconds, zoneId
ZonedDateTime zonedDateTime5 =
    ZonedDateTime.of(2015,1,30,13,59,59,999, australiaZone);
```

You can also get the current date/time from the system clock in the default time zone with:

```
ZonedDateTime now = ZonedDateTime.now();
```

From a `ZonedDateTime` you can get `LocalDate`, `LocalTime`, or a `LocalDateTime` (without the time zone part) with:

```
LocalDateTime currentDate = now.toLocalDate();
LocalDateTime currentTime = now.toLocalTime();
LocalDateTime currentDateTime = now.toLocalDateTime();
```

`ZonedDateTime` also have most of the methods of `LocalDateTime` that we reviewed in the previous chapter:

```
//To get the value of a specified field
int day = now.getDayofMonth();
int dayYear = now.getDayOfYear();
int nanos = now.getNano();
Month monthEnum = now.getMonth();
int year = now.get(ChronoField.YEAR);
long micro = now.getLong(ChronoField.MICRO_OF_DAY);
// This is new, gets the zone offset such as "-03:00"
ZoneOffset offset = now.getOffset();
// To create another instance
ZonedDateTime zdt1 = now.with(ChronoField.HOUR_OF_DAY, 10);
ZonedDateTime zdt2 = now.withSecond(49);
// Since these methods return a new instance, we can chain them!
ZonedDateTime zdt3 = now.withYear(2013).withMonth(12);

// The following two methods are specific to ZonedDateTime
// Returns a copy of the date/time with a
// different zone, retaining the instant
ZonedDateTime zdt4 = now.withZoneSameInstant(australiaZone);
// Returns a copy of this date/time with a different time zone,
// retaining the local date/time if it's valid for the new time zone
ZonedDateTime zdt5 = now.withZoneSameLocal(australiaZone);

// Adding
ZonedDateTime zdt6 = now.plusDays(4);
ZonedDateTime zdt7 = now.plusWeeks(3);
ZonedDateTime zdt8 = newYear2001.plus(2, ChronoUnit.HOURS);

// Subtracting
ZonedDateTime zdt9 = now.minusMinutes(20);
ZonedDateTime zdt10 = now.minusNanos(99999);
ZonedDateTime zdt11 = now.minus(10, ChronoUnit.SECONDS);
```

The method `toString()` returns the date/time in the format of a `LocalDateTime` followed by a `ZoneOffset`, optionally, a `ZoneId` if it is not the same as the offset, and omitting the parts with value zero:

```
// Prints 2014-09-19T00:30Z
System.out.println(
    ZonedDateTime.of(2014,9,19,0,30,0,0,ZoneId.of("Z")));
// Prints 2015-08-31T12:39:27.492-04:00[America/Montreal]
System.out.println(
    ZonedDateTime.now(ZoneId.of("America/Montreal")));
```

Daylight savings

Many countries in the world adopt what is called Daylight Saving Time (DST), the practice of advance the clock by an hour in the summer (well, not exactly in the summer in all countries but let's bear with this) when the daylight savings time starts.

When the daylight time ends, clocks are set back by an hour. This is done to make better use of natural daylight.

`ZonedDateTime` is fully aware of DST.

For an example, let's take a country where DST is fully observed, like Italy (UTC/GMT +2).

In 2015, DST started in Italy on March, 29th and ended on October, 25th. This means that on:

*March, 29 2015 at **2:00:00** A.M. clocks were turned **forward** 1 hour to*
*March, 29 2015 at **3:00:00** A.M. local daylight time instead*
(So a time like March, 29 2015 2:30:00 A.M. didn't actually exist!)

*October, 25 2015 at **3:00:00** A.M. clocks were turned **backward** 1 hour to*
*October, 25 2015 at **2:00:00** A.M. local daylight time instead*
(So a time like October, 25 2015 2:30:00 A.M. actually existed twice!)

If we create an instance of `LocalDateTime` with this date/time and print it:

```
LocalDateTime ldt = LocalDateTime.of(2015, 3, 29, 2, 30);
System.out.println(ldt);
```

The result will be:

```
2015-03-29T02:30
```

But if we create an instance of `ZonedDateTime` for Italy (notice that the format uses a city, not a country) and printed:

```
ZonedDateTime zdt = ZonedDateTime.of(
     2015, 3, 29, 2, 30, 0, 0, ZoneId.of("Europe/Rome"));
System.out.println(zdt);
```

The result will be just like in the real world when using DST:

```
2015-03-29T03:30+02:00[Europe/Rome]
```

But be careful. We have to use a regional `ZoneId`, a `ZoneOffset` won't do the trick because this class doesn't have the zone rules information to account for DST:

```
ZonedDateTime zdt1 = ZonedDateTime.of(
     2015, 3, 29, 2, 30, 0, 0, ZoneOffset.ofHours(2));
ZonedDateTime zdt2 = ZonedDateTime.of(
     2015, 3, 29, 2, 30, 0, 0, ZoneId.of("UTC+2"));
System.out.println(zdt1); System.out.println(zdt2);
```

The result will be:

```
2015-03-29T02:30+02:00[UTC+02:00] 2015-03-29T02:30+02:00
```

When DST ends, something similar happens

```
LocalDateTime ldt = LocalDateTime.of(2015, 10, 25, 3, 30);
System.out.println(ldt);
```

The result will be:

```
2015-10-25T02:30
```

When we create an instance of `ZonedDateTime` for Italy, we have to add an hour to see the effect (otherwise we we'll be creating the `ZonedDateTime` at 3:00 of the new time):

```
ZonedDateTime zdt = ZonedDateTime.of(
    2015, 10, 25, 2, 30, 0, 0, ZoneId.of("Europe/Rome"));
ZonedDateTime zdt2 = zdt.plusHours(1);
```

```
System.out.println(zdt);
System.out.println(zdt2);
```

The result will be:

```
2015-10-25T02:30+02:00[Europe/Rome] 2015-10-25T02:30+01:00[Europe/Rome]
```

We also need to be careful when adjusting the time across the DST boundary with a version of the methods `plus()` and `minus()` that takes a `TemporalAmount` implementation, in other words, a `Period` or a `Duration`. This is because both differ in their treatment of daylight savings time.

Consider one hour before the beginning of DST in Italy:

```
ZonedDateTime zdt = ZonedDateTime.of(
    2015, 3, 29, 1, 0, 0, 0, ZoneId.of("Europe/Rome"));
```

When we add a `Duration` of one day:

```
System.out.println(zdt.plus(Duration.ofDays(1)));
```

The result is:

```
2015-03-30T02:00+02:00[Europe/Rome]
```

When we add a `Period` of one day:

```
System.out.println(zdt.plus(Period.ofDays(1)));
```

The result is:

```
2015-03-30T01:00+02:00[Europe/Rome]
```

The reason is that `Period` adds a *conceptual* date, while `Duration` adds *exactly* one day (24 hours or 86, 400 seconds) and when it crosses the DST boundary, one hour is added, and the final time is *02:00* instead of *01:00*.

OffsetDateTime and OffsetTime

`OffsetDateTime` represents an object with date/time information and an offset from UTC, for example, *2015-01-01T11:30-06:00*.

You may think `Instant`, `OffsetDateTime` and `ZonedDateTime` are very much alike, after all, they all store the date and time to a nanosecond precision. However, there are subtle but important differences:

- `Instant` represents a point in time in the UTC time zone.
- `OffsetDateTime` represents a point in time with an offset (any offset).
- `ZonedDateTime` represents a point in time in a time zone (any time zone), adding full time zone rules like daylight saving time adjustments.

`OffsetTime` represents a time with an offset from UTC, for example, *11:30-06:00*. The common way to create an instance of these classes is:

```
OffsetDateTime odt = OffsetDateTime.of(
    LocalDateTime.now(), ZoneOffset.of(+03:00));
OffsetTime ot = OffsetTime.of(
    LocalTime.now(), ZoneOffset.of(-08:00));
```

Both classes have practically the same methods that their `LocalDateTime`, `ZonedDateTime`, and `LocalTime` counterparts. With an offset from UTC and without time zone variations, they always represent an exact instant in time, which may be more suited for certain types of applications (the Java documentation recommends `OffsetDateTime` when communicating with a database or in a network protocol).

Parsing and Formatting

`java.time.format.DateTimeFormatter` is the new class for parsing and formatting dates. It can be used in two ways:

- The date/time classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `OffsetDate` and `OffsetDateTime` all have the following three methods:

```
// Formats the date/time object using the specified formatter
String format(DateTimeFormatter formatter)
// Obtains an instance of a date/time object (of type T)
// from a string with a default format
static T parse(CharSequence text)
// Obtains an instance of a date/time object (of type T)
// from a string using a specific formatter
static T parse(CharSequence text, DateTimeFormatter formatter)
```

- DateTimeFormatter has the following two methods:

```
// Formats a date/time object using the formatter instance
String format(TemporalAccessor temporal)
// Parses the text producing a temporal object
TemporalAccessor parse(CharSequence text)
```

All format methods throw the runtime exception:
`java.time.DateTimeException`
All parse methods throw the runtime exception:
`java.time.format.DateTimeParseException`

`DateTimeFormatter` provides three ways to format date/time objects:

- Predefined formatters
- Locale-specific formatters
- Formatters with custom patterns

Formatter	Description	Example
`BASIC_ISO_DATE`	Date fields without separators	*20150803*
`ISO_LOCAL_DATE` `ISO_LOCAL_TIME` `ISO_LOCAL_DATETIME`	Date fields with separators	*2015-08-03* *13:40:10* *2015-08-03T13:40:10*
`ISO_OFFSET_DATE` `ISO_OFFSET_TIME` `ISO_OFFSET_DATETIME`	Date fields with separators and zone offset	*2015-08-03+07:00* *13:40:10+07:00* *2015-08-03 T13:40:10+07:00*
`ISO_ZONED_DATE_TIME`	A zoned date and time	*2015-08-03 T13:40:10 +07:00 [Asia/Bangkok]*
`ISO_DATE` `ISO_TIME`	Date or Time with or without offset	*2015-08-03+07:00* *13:40:10*

ISO_DATETIME	DateTime with ZoneId	*2015-08-03 T13:40:10+07:00 [Asia/Bangkok]*
ISO_INSTANT	Date and Time of an Instant	*2015-08-03 T13:40:10Z*
ISO_ORDINAL_DATE	Year and day of the year	*2015-200*
ISO_WEEK_DATE	Year, week and day of the week	*2015-W34-2*
RFC_1123_DATE_TIME	RFC 1123 / RFC 822 date format	*Mon, 3 Ago 2015 13:40:10 GMT*

Predefined Formatters

Style	Date	Time
SHORT	8/3/15	*1:40 PM*
MEDIUM	Aug 03, 2015	*1:40:00 PM*
LONG	August 03, 2015	*1:40:00 PM PDT*
FULL	Monday, August 03, 2015	*1:40:00 PM PDT*

Locale-specific Styles

Symbol	Meaning	Examples
G	Era	*AD; Anno Domini; A*
u	Year	*2015; 15*
y	Year of Era	*2015; 15*
D	Day of Year	*150*
M / L	Month of Year	*7; 07; Jul; July; J*
d	Day of Month	*20*
Q / q	Quarter of year	*2; 02; Q2; 2nd quarter*
Y	Week-based Year	*2015; 15*
w	Week of Week-based Year	*30*
W	Week of Month	*2*
E	Day of Week	*Tue; Tuesday; T*
e / c	Localized Day of Week	*2; 02; Tue; Tuesday; T*
F	Week of Month	*2*
a	AM/PM of Day	*AM*
h	Hour (1-12)	*10*
K	Hour (0-11)	*1*
k	Hour (1-24)	*20*
H	Hour (0-23)	*23*
m	Minute	*10*
s	Second	*11*
S	Fraction of Second	*999*
A	Milli of Day	*2345*
n	Nano of Second	*865437987*

N	Nano of Day	*12986497300*
V	Time Zone ID	*Asia/Manila; Z; -06:00*
z	Time Zone Name	*Pacific Standard Time; PST*
O	Localized Zone Offset	*GMT+4; GMT+04:00; UTC-04:00;*
X	Zone Offset ('Z' for zero)	*Z; -08; -0830; -08:30*
x	Zone Offset	*+0000; -08; -0830; -08:30*
Z	Zone Offset	*+0000; -0800; -08:00*
'	Escape for Text	
''	Single Quote	
[ ]	Optional Section Start / End	
# { }	Reserved for future use	

Custom Patterns

Assuming:

```
LocalDate ldt = LocalDate.of(2015, 1, 20);
```

These are examples of using a predefined formatter:

```
System.out.println(DateTimeFormatter.ISO_DATE.format(ldt));
System.out.println(ldt.format(DateTimeFormatter.ISO_DATE));
```

The output will be:

```
2015-01-20 2015-01-20
```

These are examples of using a localized style:

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
// With the current locale
System.out.println(formatter.format(ldt));
System.out.println(ldt.format(formatter));
// With another locale
System.out.println( formatter.withLocale(Locale.GERMAN).format(ldt));
```

One output can be:

```
20/01/15 20/01/15 20.01.15
```

And these are examples of using a custom pattern:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("QQQQ Y");
// With the current locale
System.out.println(formatter.format(ldt));
System.out.println(ldt.format(formatter));
// With another locale
System.out.println(formatter.withLocale(Locale.GERMAN).format(ldt));
```

One output can be:

```
1st quarter 2015
1st quarter 2015
1. Quartal 2015
```

If the formatter uses information that is not available, a `DateTimeException` will be thrown. For example, using a `DateTimeFormatter.ISO_OFFSET_DATE` with a `LocalDate` instance (it doesn't have offset information).

To parse a date and/or time value from a string, use one of the `parse` methods. For example:

```
// Format according to ISO-8601
String dateTimeStr1 = "2015-06-29T14:45:30";
// Custom format
String dateTimeStr2 = "2015/06/29 14:45:30";
LocalDateTime ldt = LocalDateTime.parse(dateTimeStr1);
// Using DateTimeFormatter
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
// DateTimeFormatter returns a TemporalAccessor instance
TemporalAccessor ta = formatter.parse(dateTimeStr2);
// LocalDateTime returns an instance of the same type
ldt = LocalDateTime.parse(dateTimeStr2, formatter);
```

The version of `parse()` of the date/time objects takes a string in a format according to ISO-8601, this is:

Class	Format	Example
`LocalDate`	`uuuu-MM-dd`	*2007-12-03*
`LocalTime`	`HH:mm:ss`	*10:15*
`LocalDateTime`	`uuuu-MM-dd'T'HH:mm:ss`	*2007-12-03T10:15:30*
`ZonedDateTime`	`uuuu-MM-dd'T'HH:mm:ss XXXXX[VV]`	*2011-12-03T10:15:30 +01:00[Europe/Paris]*
`OffsetDateTime`	`uuuu-MM-dd'T'HH:mm:ssXXXXX`	*2011-12-03T10:15:30 +01:00*
`OffsetTime`	`HH:mm:ssXXXXX`	*10:15:30+01:00*

If the formatter uses information that is not available or if the pattern of the format is invalid, a `DateTimeParseException` will be thrown.

Key Points

- `ZoneId`, `ZoneOffset`, `ZonedDateTime`, `OffsetDateTime`, and `OffsetTime` are the classes of the new Java Date/Time API that store information about time zones and time offsets. They are located in the `java.time` package and are immutable.
- Each time zone has an ID, represented by the class `ZoneId`. There are three types of ID.
- The first type just states the offset from UTC/GMT time. They are represented by the class `ZoneOffset` and they consist of digits starting with + or -, for example *+02:00*.
- The second type also states the offset from UTC/GMT time, but with one of the following prefixes: UTC, GMT and UT, for example, *UTC+11:00*. They are also represented by the class `ZoneOffset`.
- The third type is region based. These IDs have the format *area/city*, for example, *Europe/London*.
- If you want to create a specific `ZoneId` object use the method `of`:

```
ZoneId.of("Asia/Singapore");
```

```
ZoneId.of("+3")
```

- The first two of the above methods produce an object of type `ZoneRegion`. The last one, an object of type `ZoneOffset`.
- A `java.time.ZonedDateTime` object represents a point in time relative to a time zone.
- A `ZonedDateTime` object has three parts, a date, a time, and a time zone. It can be created with either:

```
ZoneId australiaZone = ZoneId.of("Australia/Victoria");
ZonedDateTime zonedDateTime5 =
    ZonedDateTime.of(2015,1,30,13,59,59,999, australiaZone);
```

- Or through a `LocalDate`, `LocalTime`, `LocalDateTime` or `Instant` plus a `ZoneId`.
- If we create an instance of `ZonedDateTime` for a region where a Daylight Saving Time (DST) is observed, the instance will support it, advancing the clock one hour when DST starts, and setting it back when DST ends.
- `Period` and `Duration` differ in their treatment of DST.
- `Period` adds a conceptual day to a date, while `Duration` adds an exact day to a date, without taking into account DST.
- `OffsetDateTime` represents an object with date/time information and an offset from UTC, for example, *2015-01-01T11:30-06:00*.
- `OffsetTime` represents a time with an offset from UTC, for example, *11:30-06:00*.
- `java.time.format.DateTimeFormatter` is the new class for parsing and formatting dates. It can be used in two ways:
- The date/time classes `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `OffsetDate` and `OffsetDateTime` all have the methods:

```
String format(DateTimeFormatter formatter)
static T parse(CharSequence text)
static T parse(CharSequence text, DateTimeFormatter formatter)
```

- `DateTimeFormatter` has the following two methods:

```
String format(TemporalAccessor temporal)
TemporalAccessor parse(CharSequence text)
```

- All format methods throw the runtime exception `java.time.DateTimeException`, while all parse methods throw the runtime exception `java.time.format.DateTimeParseException`.

Self Test

1. Which of the following are valid ways to create a `ZoneId` object?

A. `ZoneId.ofHours(2);`
B. `ZoneId.of("2");`
C. `ZoneId.of("-1");`
D. `ZoneId.of("America/Canada");`

2. Given:

```
ZoneOffset offset = ZoneOffset.of("Z");
System.out.println(
    offset.get(ChronoField.HOUR_OF_DAY)
);
```

Which of the following is the result of executing the above lines?

A. `0`
B. `1`
C. `12:00`
D. An exception is thrown

3. Given:

```
ZonedDateTime zdt =
    ZonedDateTime.of(2015,02,28,5,0,0,0,
                     ZoneId.of("+05:00"));
System.out.println(zdt.toLocalTime());
```

Assuming a local time zone of +2:00, which of the following is the result of executing the above lines?

A. `05:00`
B. `17:00`
C. `02:00`
D. `03:00`

4. Given:

```
ZonedDateTime zdt =
    ZonedDateTime.of(2015,10,4,0,0,0,0,
                     ZoneId.of("America/Asuncion"))
    .plus(Duration.ofHours(1));
System.out.println(zdt);
```

Assuming that DST starts on October, 4, 2015 at 0:00:00, which of the following is the result of executing the above lines?

A. `2015-10-04T00:00-03:00[America/Asuncion]`
B. `2015-10-04T01:00-03:00[America/Asuncion]`
C. `2015-10-04T02:00-03:00[America/Asuncion]`
D. `2015-10-03T23:00-03:00[America/Asuncion]`

5. Given:

```
ZonedDateTime zdt =
    ZonedDateTime.of(2015,3,22,0,0,0,0,
                     ZoneId.of("America/Asuncion"))
        .minus(Period.ofDays(1));
System.out.println(zdt);
```

Assuming that DST ends on March, 22, 2015 at 0:00:00, which of the following is the result of executing the above lines?

A. `2015-03-21T01:00-03:00[America/Asuncion]`
B. `2015-03-21T00:00-03:00[America/Asuncion]`
C. `2015-03-20T23:00-03:00[America/Asuncion]`
D. `2015-03-21T02:00-03:00[America/Asuncion]`

6. Which of the following statements are true?

A. `java.time.ZoneOffset` is a subclass of `java.time.ZoneId`.
B. `java.time.Instant` can be obtained from `java.time.ZonedDateTime`.
C. `java.time.ZoneOffset` can manage DST.
D. `java.time.OffsetDateTime` represents a point in time in the UTC time zone.

7. Given:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(
    formatter
    .withLocale(Locale.ENGLISH)
    .format(LocalDateTime.of(2015, 5, 7, 16, 0))
);
```

Which of the following is the result of executing the above lines?

A. `5/7/15 4:00 PM`
B. `5/7/15`
C. `4:00 PM`
D. `4:00:00 PM`

8. Given:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("HH:mm:ss X");
OffsetDateTime odt =
    OffsetDateTime.parse("11:50:20 Z", formatter);
```

Which of the following statements is true about the above lines?

A. The pattern *HH:mm:ss X* is invalid.
B. An `OffsetDateTime` is created successfully.
C. *Z* is an invalid offset.
D. An exception is thrown at runtime.

Open answers page

---

21. Core Date/Time Classes 23. Java I/O Fundamentals